Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach

Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach

Josie E. Rodriguez Condia†, Matteo Sonza Reorda‡,
Politecnico di Torino, Dept. of Control and Computer Engineering, Torino, Italy
{†josie.rodriguez, ‡matteo.sonzareorda}@polito.it

Abstract [1] **—In the last decade, General Purpose Graphics Processing Units (GPGPUs) have been widely employed in high demanding data processing applications including multimedia and high-performance computing due to their parallel processing capabilities. Nowadays, these devices are considered as promising solutions also for high-performance safety-critical applications, such as autonomous and semi-autonomous vehicles. Current GPGPUs are designed targeting challenging execution requirements, e.g., related to performance and power constraints, forcing designers to use aggressive technology scaling solutions. Nevertheless, some implementation technologies are prone to introduce faults in the device during the operative life adding unaffordable effects and errors for the safety-critical domain. Hence, effective in-field test solutions are required to guarantee the target reliability levels. In this paper, we propose in-field test solutions based on Software-Based Self-Test (SBST) targeting the control-path of pipeline registers located in the Streaming Multiprocessor (SM) of a GPGPU. We resort to a multiple-kernel approach to detect permanent faults in these register fields. The solutions were designed employing NVIDIA CUDA, when possible, and lower level constructs elsewhere. Several usages and compilation restrictions are also described. Fault simulation results on an open-source VHDL GPGPU (FlexGrip) implementation of the G80 architecture of NVIDIA are reported, showing the effectiveness and limitations of the approach.**

Keywords— ***fault simulation, functional testing, GPGPUs, pipeline registers, SBST.***

I. INTRODUCTION

General Purpose Graphics Processing Units (GPGPUs) have been used in the last years in highly demanding applications to process large amounts of data including multimedia and high-performance computing. Nowadays, these devices are considered as feasible solutions also in new complex and safety-critical applications, such as autonomous and semi-autonomous cars [1]. These devices are designed following execution requirements, including performance and power constraints, forcing designers to employ aggressive technology scaling solutions. Moreover, it has been shown that some implementation technologies are more prone to introduce faults (permanent or transient) during the operative life of the device [2, 3] causing unaffordable failures in the safety-critical domain. Unfortunately, the architectural complexity of GPGPUs aggravates the design of effective and affordable test techniques oriented to verify the integrity of internal modules during in-field operation [4].

In the industry, testing solutions for complex embedded systems are based on Design for Testability (DfT) approaches, such as Built-In Self-Test (BIST), and functional test. DfT is effective for end of manufacturing test. However, new reliability challenges exist in GPGPU devices, and DfT structures are not commonly accessible during in-field operation. On the other hand, functional test methods based on Software-Based Self-Test (SBST) employ the available modules in the device to perform the test. In SBST, a set of instructions is employed to generate test patterns on target modules and detect faults or verify the functionality of them, thus, checking the internal structures of the device.

A GPGPU is mainly composed of groups of Streaming Multiprocessors (SMs). Each SM includes multiple levels of pipeline stages to improve application performance. A register is placed between each pair of adjacent stages to temporarily store information about the multiple instructions executed in the stage. These registers are crucial for the SM operation and stores temporary information about the data and control signals employed in the different stages of the SM. A fault affecting these structures could generate critical and unexpected behaviors, from an erroneous result, if the affected location is a data-path register field, until a system crash, when a control-path register field is affected. Hence, faults in these structures could often be unacceptable for safety-critical applications. In particular, faults in control-path fields of Pipeline Registers (PRs) are complex to detect during the device operation. Moreover, systematic solutions for in-field test of faults in these GPGPU structures are still missing.

Some works [5-7] proposed tools to inject soft-errors at the low-level code in real o micro-architectural models of GPGPUs, in particular, to support the analysis of transient faults effects. However, the injection locations are limited to some data-path units and it is not possible to determine a Fault Coverage (FC) for permanent faults resorting to these solutions. In other works, the authors employ RT-level models and real GPGPUs to propose test techniques targeting permanent faults in some modules, such as register files (RF) [8], memories [9] and controllers [10]. Similarly, other works [11, 12] analyzed the effect of transient faults on data-path and found a relation among the fault effects, the employed instructions and the module usage. In [13], the authors proposed hardening techniques to mitigate the effect of transient faults in the PRs calculating the effects of faults affecting each register for some selected applications. However, to the best of our knowledge, there are no works proposing methods addressing permanent

[1] This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 72232.

fault detection in GPGPU PRs and reporting experimental data on their effectiveness.

In this paper, we analyze the feasibility of using Software-Based Self-Test (SBST) techniques to detect permanent faults in the PRs in the Streaming Multiprocessors (SMs) of a GPGPU. This work targets the control-path fields in the pipeline registers considering that these structures are present in GPGPU technologies and include control and management information. This information is crucial for SM operation and instruction execution. Moreover, these are not easy testable and are located across PRs in the SM. We neglected data-path fields since it is known that the test of data-path structures is more dependent on low-level implementation details and can be successfully achieved using techniques such as the one proposed in [14].

We resort to a multiple-kernel approach targeting different sub-sections of the control-path fields in the PRs, showing that traditional in-field functional techniques developed for CPUs can be applied in this case, provided that a careful combination of high- and low-level programming structures are adopted. Experiments were performed employing fault simulation on an open-source VHDL GPGPU (FlexGrip) implementation of the G80 architecture of NVIDIA [15]. The NVIDIA's terminology is employed to describe the proposed techniques. However, it should be noted that these methods can be adapted to other GPGPU technologies.

Results show the effectiveness and limitations of the approach. As far as we know, this is the first work presenting an experimental evaluation (i.e., assessing the achieved FC) of SBST approaches to detect permanent faults in the PRs of a GPGPU.

The paper is organized as follows: Section II briefly introduces the basic architecture of a GPGPU, the pipeline registers in the FlexGrip model and some operative restrictions of FlexGrip. Section III introduces the proposed SBST techniques to detect permanent faults in the pipeline registers. Section IV describes the fault injection environment. Section V reports some experimental results and Section VI finally draws some conclusions.

II. BACKGROUND

GPGPUs are special purpose parallel processors designed to process simultaneously multiple tasks in groups (*32 threads or a warp*) using SMs. Each SM includes execution units (Scalar Processors, or SPs), caches, (shared) memories, RFs, a task scheduler, and a dispatcher controller. The SM executes the same instruction (warp instruction) on different SPs using particular thread operands. Internally, the SM employs multiple pipelines stages to process the warp instruction and improve performance. The next section describes the pipeline register structures in the FlexGrip GPGPU.

A. Pipeline registers in FlexGrip

These registers are placed between every pair of pipeline stages to store temporary information from the previous stage and supply information into the next one. In FlexGrip, PRs are distributed between the five stages in the SM, named *Fetch*, *Decode*, *Read*, *Execution* and *Write-back*. An additional stage is considered, *Warp*, which is the interconnection stage between the Warp scheduler and the Fetch stage (see Figure 1).

Every pipeline stage partially manages the execution of warp instructions in the SM. The execution starts with the *Warp* scheduler dispatching a warp to be executed in the SM. The *Fetch* stage translates the warp program counter into the instructions to operate. The *Decode* stage converts the instruction operational code into memory or register locations, including operands and destinations, and instructions formats. The *Read* stage collects the different operators and assigns data formats for the active threads. The *Execution* stage executes control-flow or logic-arithmetical operations depending on predicate conditions and input format parameters. The *Write-back* stage stores result in the target memory or register locations. Finally, the Warp scheduler checks the instruction operation and dispatches a new instruction for the next cycle.

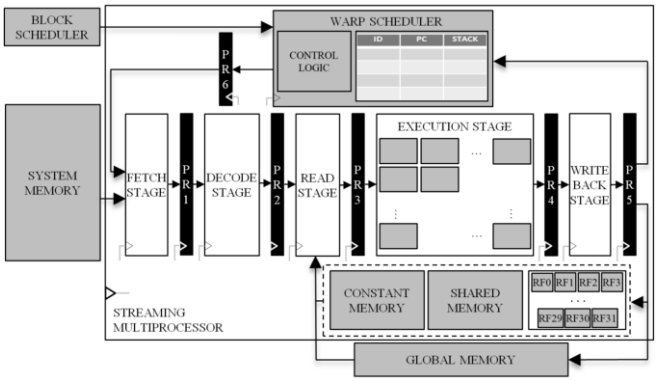

FIG 1. THE GENERAL SCHEME OF THE SM IN THE FLEXGRIP GPGPU

The PRs store mainly operands for warp instruction execution. Nevertheless, these also include control information related to the warp instruction status. In the *Warp-Fetch* (W-F) registers, these are composed of control fields related to the actual instruction warp status and execution on the SM including the Warp program counter (WPC), the initial and active thread mask (AThM), parameters for shared memory and general purpose registers size configuration. The PR in the *Fetch-Decode* (F-D) stage includes the same information of the previous stage, adding the warp instruction operational code. The *Decode-Read* (D-R) PR stores the format of the specific instructions fields to activate some operational modes or sub-modules in the next stage. The *Read-Execution* (R-E) PR additionally includes the Temporary Registers (TRs), which handle operands and predicate conditions for each SP in the execution stage. The *Execution-Writeback* (E-Wr) PR also contains some TRs. However, they store the result or partial result of a warp instruction. Table 1 summarizes the basic information about the control-path fields of PRs.

TABLE 1. GENERAL INFORMATION ABOUT PRS IN FLEXGRIP (CONTROL-PATH ONLY).

Regs	Warp Instructions	Warp Status	Instruction Opcode	Instruction Formats	Bits per Reg
F-D	X	X			237
D-R	X	X	X	X	391
R-E	X	X		X	302
E-Wr	X	X		X	251
Wr-W	X	X			133
W-F	X	X			140

The high number of bits per register in the *R-E* and *E-Wr* registers is caused by the large number of registers employed in the TRs to handle the operands for each SP. These structures temporary store operands and results of logical, arithmetical and control-flow operations of each thread on an SP in the SM. TRs are organized in sets, one per SP. Each register set is composed of 6 groups of registers and each group includes four 32-bit registers. The group of registers 0, 2 and 4 in the *R-E* register store the operands (SRC1, SRC2, and SRC3). Similarly, Group 0 in *E-Wr* register stores the result (DST).

B. FlexGrip restrictions

The work reported in this paper has been performed on a modified version of the original Flexgrip model described in [15], where we fixed some bugs, removed some restrictions and added some extensions. There are still some operational

restrictions in the adopted Flexgrip model related to the programming environment. Those are: *i)* FlexGrip executes one kernel per time. In order to launch other kernels, it is required to load memories and configuration parameters. *ii)* The shared memory and RF parameters are programmed during the configuration stage. ***iii)*** Flexgrip supports 27 instructions in 78 formats, only. Moreover, it does not provide any floating point support. *iv)* The CUDA compiler may generate unsupported instructions for FlexGrip, and the optimizations may change the type and order of the instruction, thus possibly creating code which cannot be executed by FlexGrip.

III. PROPOSED SBST METHOD

We adopted a bottom-up approach designing multiple programs (kernels) to test permanent faults in the control-path fields of the GPGPU pipeline registers. Each kernel focuses on some specific parts of the register fields. In the end, the cumulative FC achieved by all the kernels is assessed. Each kernel is written through a high-level compiler (CUDA) when possible. Moreover, we added some assembly instructions (*SASS*) if required. It is worth noting that SASS assembly language is not fully known as it has not been released by NVIDIA. In FlexGrip, a file generation description was added in order to represent the global memory results after kernel termination.

A. Proposed functional test methods

PRs are divided into two main groups and multiple sub-sets for the purpose of developing multiple functional test methods, one for each target. Fig 2 represents the pipeline fields division. The proposed test methods are explained in the following sections.

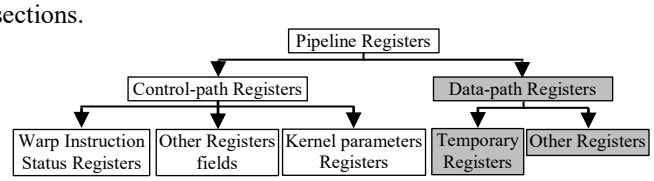

FIG. 2. DIVISION OF PIPELINE REGISTER FIELDS

B. Algorithms to test the warp instruction status registers

As shown in Table 1, each PR stores warp instruction and status information across the SM. It means that, when targeting one sub-set of registers in one PR, it is enough to generate the detection on other PRs. The test method for the WPC field and AThM are introduced in the next subsections.

1) WPC technique

This method (*PC_T*) employs a main program which calls a set of sub-routines, strategically placed in memory, in order to generate test patterns in the WPC registers of each PR. Each sub-routine is composed of a Signature-per-Thread (SpT) and a Counter-per-Thread (CpT). These two elements increase the observability of the target registers in the memory of the test program and also stop the execution if a permanent fault affects one of these fields.

The CpT verifies if a fault generates loop conditions and a hanging effect in the system. Kernel termination instructions (*RET*) are placed in memory locations between two subroutines to solve this issue. These instructions stop the kernel execution when a control-flow instruction does not reach a target memory location due to a fault turning hanging conditions into fault detections. Each subroutine checks the CpT value. If this value corresponds to the expected one, the SpT is loaded, actualized and stored. Then, a new subroutine is launched. Otherwise, the kernel is stopped. Fig 3 shows the operations performed by a subroutine.

In the *PC_T* implementation, it was necessary to replace the *CALL* and *RETURN* instructions, not supported by Flexgrip, with unconditional and control-flow instructions (*BRA*). Thus, the test program consists of multiple unconditional jumps to and from subroutines. The lower bits in WPC registers were not explicitly tested with subroutines considering that instructions in master program implicitly generate patterns for them.

The program kernel is configured with 32 TpB and one BpG. The execution of one warp checks the state of the WPC fields, considering that WPC is shared for all threads in a warp. The program kernel was designed to skip thread divergence and avoid the incidence of other modules during execution.

```
RET                                   (*)  ► Added before starting the routine
Init:  CpT_S ← Expected_param              ► Load expected CpT in subroutine
       load_CpT(i);                        ► Load CpT from global memory
       If CpT[i] == CpT_S then             ► Compare CpT and CpT_S
         CpT[i] ← CpT[i] + 1               ► Actualize the CpT
         Store_CpT(i);                     ► Store the CpT in Global memory
         load_SpT(i);                      ► Load the SpT from global memory
         SpT[i] ← SpT[i] + S_Param         ► Actualize the SpT
         Store_SpT(i);                     ► Store the SpT in Global memory
       else                                ►
         RET                           (*) ► Finish kernel Execution
```

FIG 3. PSEUDO-CODE OF THE SUBROUTINE TARGETING THE WPC FIELDS. (*) ADDED ASSEMBLY INSTRUCTIONS. SUBROUTINES WERE EXPLICITLY PLACED IN THE SYSTEM MEMORY.

2) AThM technique

This technique is an adaptation of the M3 algorithm, introduced in [10]. This was originally intended to detect permanent faults in entry-lines of the warp status memory in the warp scheduler of GPGPUs. The same warp status information is presented in the warp status memory and pipeline registers. Thus, is possible to adapt this method targeting AThM fields. The method generates thread divergence operations to supply test patterns targeting the AThM fields. Two control-flow instructions are employed to start and finish the divergence in the model. The first instruction, a conditional control–flow type, is activated through logical comparisons between the Thread Identifier and a set of constant values. These values are all the potential Thread Identifiers in a warp (0-31). The divergence generates two potential paths (*Taken* and *not-taken*). In the first one (*Taken*), an SpT is actualized and stored in memory. The other path (*not-taken*) is not employed and an unconditional control-flow instruction returns to the convergence point for a new comparison. In the end, 32 comparisons are performed in a sequential fashion and each bit register is tested. The SpT is employed as observability mechanism of a fault in memory. Fig 4 shows the general procedure to test AThM fields.

```
Load_ThreadId();                          ► Load the Thread.Id
Load_SpT(i);                              ► Load signature
for i ϵ {set of ThreadId in a warp} do    ► Evaluate for every Thread.Id
   j ← Params[k];                         ► Load the comparison parameter
   if i == j then                         ► Comparison of Thread.Id and Constant
      Load_SpT(i);                        ► Load SpT from global memory
      SpT [i] ← SpT [i] + 1               ► Set signature
      Store_SpT(i);                       ► Store SpT to global memory
   else
      NOP                           (*)   ► Not used path
   k ←k+1                                 ► Change constant value (Convergence Point)
```

FIG. 4. PSEUDO-CODE OF THE TEST PROGRAM FOR THE ATHM FIELDS. (*)ADDED ASSEMBLY INSTRUCTIONS.

We proposed two solutions by changing one logic operation in order to select between detection and diagnosis test. In the first case, a fast fault detection test is designed with logical comparison through an AND operator. On the other hand, XOR operators are used in the diagnosis test version.

The diagnosis test (*WS_T_D*) is able to identify an individual permanent fault in the AThM field. On the other hand, a detection test requires only two comparisons. For this purpose, two variations where proposed. In the first (*WS_T_V1*),

the divergence is performed on even and odd thread groups (*16 threads per time*). Initially, an even constant is loaded and the comparison generates divergence on even threads. Those threads actualize the SpT and return to convergence. Then, an odd constant is loaded and the previous process is repeated with the odd threads. Finally, a comparison of SpTs in memory is performed to detect a permanent fault. The (*WS_T_V2*) test version employs only one comparison to generate the divergence (an even constant parameter). Nevertheless, in both paths (*even* and *odd*) the SpTs are actualized. All test programs are configured with 32 TpB and one BpG taking into account that the execution of one warp in the SM is enough to test the fields in the pipeline registers.

C. Method to test the kernel parameters fields

1) The GPRS size

These fields define the number of registers to be employed for each thread during kernel execution and are programmed during the device configuration stage. Thus, one kernel is not able to generate the required test patterns. The proposed method is based on designing three program kernels forcing the compiler to use an expected number of registers and generate the patterns.

Test kernels *GPR_T_3R*, *GPR_T_12R* and *GPR_T_63R* were designed using 3 (0x03), 12 (0x0C) and 63 (0x3F) registers, respectively, which are also the patterns for the target field. The pattern selection followed the guidelines of the CUDA Tool-kit manual. The GPGPUs with computer capability 1.0 is able to handle up to 63 registers per thread. Over this limit, the compiler generates optimizations or data transfer to other memories.

GPR_T_3R program executes one logical and one arithmetical operation. This is configured with 1024 TpB and one BpG in order to use a complete SM. *GPR_T_12R* kernel executes a set of addition operations on global memory locations. This program is configured with 64 TpB and one BpG. Considering that RF placement policy assigns the registers of each thread in a consecutive way, it is possible to detect faults with this configuration. Finally, *GPR_T_63* kernel computes an accumulative addition using each register as part of the result avoiding the optimization by the CUDA compiler. The kernel is configured with 256 TpB and 1 BpG. Each test program includes an SpT. Kernel termination and SpTs are used as observability mechanism for fault detection.

GPR_T_3R is employed to test permanent faults in "1" on the higher part of the register field. Similarly, *GPR_T_12R* detects those in the higher part and the lower part of the field. *GPR_T_63R* is employed to test permanent faults in "0", considering that a fault would overlap other thread registers, thus, corrupting the result. *GPR_T_3R* kernel was also able to generate patterns to test other control-path fields by employing a large number of threads in the kernel.

2) GPRS and shared memory base fields.

These fields are also programmed during the device configuration stage. Nevertheless, the execution of multiple blocks or a large set of threads per block, in the same SM, is able to generate test patterns on these fields. The proposed approach is a combination of both approaches.

GPR_T_3R kernel is reused to test the low part of the target fields, considering that it uses the maximum number of threads per block and a low number of registers. On the other hand, the high part requires the assignation of distributed memory addresses across the RF. For this purpose, we designed one kernel (*B_T*) employing 16 registers per thread and configured with 8 BpG. In this way, the 9 bits in the GPRS base field can be tested. The test kernels execute a set of arithmetical operations in order to employ the selected number of registers; finally, the SpT is stored in global memory.

D. Other register fields.

This kernel targets the missing register fields in the control-path fields. Most of them are presented in *D-R* register and are composed of the instructions op-codes, predicate registers flags, immediate operands values, and logical and arithmetic selectors. It was considered to employ a pseudo-random kernel employing most instructions, thus generating most test patterns. Nevertheless, this solution is feasible only when the ISA of the GPGPU is well known and it is directly generated at the assembly level. Nevertheless, CUDA employs multiple compiler optimizations removing instructions or modifying those that do not contribute to a thread result in memory. This restriction minimizes the effectiveness of this method.

As a solution, this kernel employs most representative instruction op-code to increase missing FC through selective operations. Then kernel (*PSR_T*) is designed to generate the highest number of potential variations on some selected target fields. Those fields are: *i)* operand order and sign, *ii)* immediate operand parameters, *iii)* predicate flags and *iv)* op-code, which represent a high percentage of the missing faults.

Targets (*i* and *ii*) require the generation of multiple arithmetic operations. On the other hand, the pattern for (*iii)* requires the explicit comparison of parameters. In order to avoid the compilation optimizations and force it to generate the expected pattern in those fields, the comparisons are made based on memory locations. An initial approach considered seven comparisons (*!=*, <, ==, <=, >, < | >, >=) to generate the patterns considering an unknown op-code. In the optimized version, it was required only two comparisons.

The op-code generation was carried out employing memory and kernel parameters movement combined with shift operations. Those instructions were analyzed following the CUDA compilation and analyzing the assembly code of multiple arithmetics and movement operations. The kernel is configured with 32 TpB and one TpG, since SM shares those fields during the execution of a warp.

E. Compiler restrictions in kernel implementation

In some of the proposed methods, problems and restrictions were faced during kernel implementation. Those restrictions are caused by the CUDA environment, which employs advanced algorithms for resource reduction and performance improvement in the application.

In the *PC_T* program, the RET instructions were manually added in free memory locations to terminate the program execution, since; CUDA compiler removes all instructions without any direct relation with the kernel execution. Besides, each subroutine was placed in the target location. Similarly, in *WS_T_x* kernels, the implementation required the explicit comparison of each thread identifier with the constant parameter independently in order to generate the expected divergence.

In the description, the *GPR_T_xR* kernels avoid arrays and matrices definitions. In these kernels, the register declaration is replaced with an independent declaration of each variable. A consecutive register declaration, such as an array would be interpreted by the compiler as local memory locations for the kernel, so limiting the target of the test kernel. Moreover, the command to increase the registers usage per thread was required in order to guarantee the total of registers employed in the test kernel. Finally, every register, in a thread, should be part of a

memory store operation in order to avoid optimizations and reduce the number of registers employed.

IV. FAULT INJECTION ENVIRONMENT

The environment was developed based on a high-level controller described in *Python* language. This controller translates a fault location into the command sequence for the simulator hosting the model (*ModelSim*), following the guidelines introduced in [16]. The tool is composed of a fault controller, a fault decoder and a fault checker and classifier. This framework also employs a multi-thread fault simulation approach [17, 18] to increase fault simulation performance. This method is implemented by dividing the fault list in equal size chunks and launching different faults campaigns, each working on a fault list chunk. 10 chunks were employed per fault campaign.

A fault injection campaign starts by defining and sending the fault list. This list includes all faults locations and the selected fault model (stuck-at faults model). Then, the fault controller performs a fault-free (*golden*) simulation and the memory results and kernel time simulation are stored as reference parameters during the fault campaign. The fault decoder reads one line from the fault list and translates it into the command sequence for Modelsim. This command is applied and the fault simulation starts. The maximum fault execution time is fixed at twice the golden execution time in order to consider potential performance degradation effects by the fault.

The fault checker and classifier compares the memory results and execution time to classify the fault. In the tool, the faults are classified as *i) Silent Data Corruption fault* (SDC), when the fault generates mismatches in memory results, *ii) Hanging (Crash) fault*, if the fault stops or prevents the kernel execution, *iii) Timeout*, if the fault affects the system introducing a delay in the kernel execution while the results are not affected, and *iv) Silent*, when the fault does not affect the system execution and results.

The fault campaign starts again by reading another line from the fault list. In the end, two files are created describing the effect of every fault in the system and the total classification of faults. One fault simulation was performed per injected fault.

V. EXPERIMENTAL RESULTS

In the experiments, we injected 2,382 faults in the control-path fields of the PRs. For the purpose of this paper, we only considered the RT-level model of the GPGPU: hence, we limited our analysis to the stuck-at faults on the inputs and outputs of the Flip-Flops composing each register. FlexGrip was configured with one SM and 32 SP-cores in the SM during the fault injection campaigns. Fault simulation campaigns required about 6 hours to be completed. The experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GB of RAM.

We performed injection campaigns on four representative benchmarks to compare the FC of applications with the one provided by the proposed solutions. The selected applications are: an embarrassingly parallel operation (*Vector_Add*) performing the addition of two arrays of 64 elements, the matrix multiplication of two (8x8) matrices, the *FFT* of a signal with 64 elements and the *Edge Detection*, based on the Sobel algorithm with a (5x5) stencil element applied to an (16x16) image.

Table 2 shows the characteristics of the developed SBST programs and the four applications. It shows that most of the kernels are composed of a low number of instructions and have a short execution time. Tables 3 and 4 show the achieved FC in the targeted structures. The total FC does not consider functionally untestable faults (FUFs) in the system, i.e., faults that cannot be tested resorting to a functional approach. For the identification of FUFs we adopted a method derived from the one presented in [19]; unfortunately, the identification of all FUFs in a complex circuit goes beyond the state of the art techniques. For the considered applications, Table 3 presents the average result of multiple simulations employing various data input sets. Results show that benchmarks provide a relatively moderate FC (32-57%). Moreover, a high percentage of fault effects are detected through hanging conditions in the system, stopping the operative state of the device.

TABLE 2. CHARACTERISTICS OF THE IMPLEMENTED TEST KERNELS AND APPLICATIONS. (*) USING CPT

SBST kernels or Benchmark	Execution time (Clock Cycles)	Memory size (Bytes)
WS_T_D	16,449	128
WS_T_V1	2,175	128
WS_T_V2	1,913	128
TR_T	2,273	384
GPR_T_3R	23,586	8,192
GPR_T_12R	103,930	400
GPR_T_63R	283,714	1500
PC_T	31,570	128 / 256(*)
B_T	178,750	9,256
PSR_T	7,313	2,304
VectorAdd	28,565	768
MatrixMul	201,365	768
Edge Detection	688,305	2048
FFT	584,265	512

TABLE 3. FC OF SELECTED APPLICATIONS IN CONTROL-PATH FIELDS OF PRS

Kernel	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
VectorAdd	18.10	20.82	0.62	32.37	**39.54**
MatrixMul	9.74	42.67	0.92	43.66	**53.33**
Edge Detection	19.89	49.44	1.03	57.60	**70.36**
FFT	21.89	42.36	0.67	53.15	**64.92**

TABLE 4. FC OF THE PROPOSED SBST APPROACH

Kernel	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
WS_T_D	4.61	25.23	16.67	38.08	43.51
WS_T_V1	4.77	23.33	13.85	34.34	41.95
WS_T_V2	4.82	23.64	13.95	34.72	42.41
GPR_T_3R	14.51	21.85	0.82	30.35	37.07
GPR_T_12R	16.77	21.49	0.51	31.74	38.77
GPR_T_63R	20.10	22.49	0.56	35.10	42.87
B_T	9.13	22.51	1.23	26.91	32.87
PC_T	21.69	17.59	0.41	38.37	39.69
PSR_T	19.74	23.54	4.46	39.08	47.74
Overall	38.31	23.44	18.51	65.70	**80.26**

VectorAdd and *MatrixMul* applications employ mainly data-path structures including data-path fields in the PRs of GPGPUs. However, the execution is affected by the incidence of a fault in the control-path fields. In the first application, faults are distributed among a system hanging condition and an SDC in results. In contrast, most faults in *MatrixMul* generate hanging conditions, due to the execution of flow-control and conditional instructions. Similarly, *FFT* and *Edge Detection* kernels, which are composed of multiple control-flow instructions, are more prone to fault effects belonging to the system hanging category.

Every proposed SBST kernel achieves a low FC (lower than 40%). Nevertheless, as explained below, the multi-kernel test approach is composed of multiple kernels, designed to target different pipeline registers fields, and executed independently. The FC in the control path is obtained as the cumulative number of faults detected by all the test kernels. The joint testable FC of those kernels reaches a relatively high percentage (80% in control-path). Moreover, the multi-kernel SBST approach is able to detect up to 38.31% of the permanent faults employing only memory results, a traditional mechanism for in-field test. On the other hand, the benchmarks are only able to detect 21%

of the faults with the same detection mechanism, showing the effectiveness of the SBST approach.

Edge detection is able to detect 70% of the permanent faults in the control-path field of the PRs. Nevertheless, 49.4% of it is through hanging detection. On the other hand, the proposed kernels reduced in up to 26% the hanging conditions and translating them into memory errors.

The multi-kernel approach also guarantees that the in-field test can be performed employing chunks (multiple kernels) with short execution time. In Table 4, we reported both the total and the Testable FC%. The total FC has been computed excluding FUFs. Multiple FUFs can be found in the control-path of the GPGPU. These include faults affecting the two lowest bits of each WPC pipeline register, the initial active thread mask, and some other fields that are present in the design but did not affect the benchmarks or the SBST kernels execution. In the end, 456 faults in the control-path are labeled as FUFs.

Table 5 reports the FC in the control-path fields for each PR using the proposed multi-kernel approach. The proposed method seems to be globally effective for fault detection on most of the fields in the pipeline registers.

TABLE 5. FAULT COVERAGE IN THE PIPELINE REGISTERS (PRS) STAGES

Pipeline Register	SDC (%)	Hanging (%)	Timeout (%)	Total FC (%)	Testable FC (%)
F-D	51.24	16.17	9.20	64.98	76.62
D-R	27.89	8.45	6.06	38.49	42.39
R-E	36.21	25.0	19.82	46.69	81.03
E-Wr	33.96	14.71	18.45	50	67.11
Wr-W	45.36	28.35	16.49	65.79	90.21
W-F	50.48	25.48	15.38	68.21	91.83
Overall	38.31	23.44	18.51	65.70	**80.26**

The relatively low FC in some PRs (*D-R*, and *E-Wr*) is mainly caused by restrictions stemming from the adoption of a high-level kernel description and implementation using the CUDA compilation tool. This tool sometimes removes or changes the execution order, instruction type, and operand placement in the device depending on the program description, the compiler configuration, and the optimizations options. This behavior is intended to increase the execution performance in the device. Nevertheless, from the reliability viewpoint, this abstraction level introduces restrictions for test pattern generation in fields, such as instruction op-codes, special operand types, physical memory addresses and fields that depend on external configuration units such as the block scheduler. Most previous registers fields are found in the *D-R* and *E-Wr* PRs. A naïve solution to improve the FC is the addition of assembly instructions, as we did in some of the proposed methods and increasing by up to 10% the obtained FC. However, this solution is limited by the incomplete knowledge of the SASS specifications as released by NVIDIA.

It is worth noting that, the proposed approaches target the fault detection employing the SDC mechanisms (i.e., looking at the memory content). This effect can be observed in all PRs results. Moreover, the proposed methods were partially effective in detecting faults in the targeted data-path fields by checking memory errors without affecting the system operation.

VI. CONCLUSIONS

Thanks to the availability of an improved RT-level model of a GPGPU, we could for the first time assess the Fault Coverage of SBST programs on the pipeline registers of the internal SPs. A multi-kernel test approach composed of multiple SBST programs was proposed to test the control-path fields in the pipeline registers. Resorting to an RT-level fault simulation environment we could compute the related FC, which amounts to more than 80%. The test kernel implementation was based on a combination of a high-level environment (CUDA) and an assembly level tool to add suitable SASS instructions. This work revealed and detailed multiple compiler restrictions and constraints during test kernel implementation at high-level. Each proposed SBST kernel targeted different portions of the control-path fields in the pipeline register. An overlap of the proposed solutions is able to detect a considerable percentage of faults employing only memory results comparisons as the detection mechanism.

We are currently working to further improve the proposed SBST methods, to assess which of the still untested faults are functionally untestable, and to extend the work to other GPGPU units.

REFERENCES

[1] W. Shi, *et al.*, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration,* vol. 59, pp. 148-156, 2017/09/01/ 2017.

[2] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 129-134.

[3] E. Ibe, *et al.*, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," *IEEE Transactions on Electron Devices,* vol. 57, pp. 1527-1538, 2010.

[4] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, *et al.*, "GPGPUs: How to combine high computational power with high reliability," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1-9.

[5] S. K. S. Hari, *et al*, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 249-258.

[6] S. Tselonis, *et al*, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 90-100.

[7] N. Farazmand, *et al.*, "Statistical fault injection-based AVF analysis of a GPU architecture," *Proceedings of SELSE,* vol. 12, 2012.

[8] S. Di Carlo, *et al.*, "A software-based self test of CUDA Fermi GPUs," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1-6.

[9] S. Di Carlo, *et al.*, "Increasing the robustness of CUDA Fermi GPU-based systems," in *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, 2013, pp. 234-235.

[10] B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "About the functional test of the GPGPU scheduler," In *On-Line Testing Symposium (IOLTS) 2018 IEEE 24th International*, Platja d'Aro, Costa Brava, Spain, 2018.

[11] M. Gonçalves, M. Saquetti, F. Kastensmidt, and J. R. Azambuja, "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files," *Microelectronics Reliability,* vol. 76-77, pp. 665-669, 2017/09/01/ 2017.

[12] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs parallelism management on safety-critical and HPC applications reliability," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 2014, pp. 455-466.

[13] M. Gonçalves, M. Saquetti, and J. R. Azambuja, "Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques," *Microelectronics Reliability,* vol. 88, pp. 931-935, 2018.

[14] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *2006 IEEE International Test Conference*, 2006, pp. 1-9.

[15] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230-237.

[16] W. Nedel, F. L. Kastensmidt, and J. R. Azambuja, "Evaluating the effects of single event upsets in soft-core GPGPUs," in *Test Symposium (LATS), 2016 17th Latin-American*, 2016, pp. 93-98.

[17] D. Alexandrescu, "Circuit and System Level Single-Event Effects Modeling and Simulation," in *Soft Errors in Modern Electronic Systems*, ed: Springer, 2011, pp. 103-140.

[18] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.,* vol. 1, pp. 171-186, 2004.

[19] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, and M. Sonza Reorda, "About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, 2018, pp. 1-6.